Universität Augsburg

Decomposing Balsa-STGs (Working Notes)

Stanislavs Golubcovs, Walter Vogler

Report 2014-01 April 2014

Institut für Informatik
D-86135 Augsburg

Decomposing Balsa-STGs (Working Notes)

Stanislavs Golubcovs, Walter Vogler
Institut für Informatik, Universität Augsburg, Germany

Abstract

The DFG-project 'Optacon' is concerned with the resynthesis of speed-independentcircuits using STGs (a variant of Petri nets). One main issue is to decompose a large STG specifying the desired circuit behaviour into a collection of components that can be synthesized separately and together implement the specification. This report collects a number of working notes regarding useful decomposition; it assumes acquaintance with the topic.[1]

1 Motivation

Asynchronous circuits are difficult to design due to their inherent complexity. In the Balsa approach [1, 2], a speed-independent circuit can be specified in the high-level hardware specification language BALSA and synthesized using syntax-directed translation. This makes design much easier, but the performance of the resulting circuits is still not good enough. Resynthesis has been suggested to improve this situation: the idea is to translate the whole or parts of a control path generated by Balsa into an STG (a Petri net, where the transitions are labelled with signal edges) and to generate a more efficient circuit with logic synthesis, using tools like MPSAT and Petrify.

One problem with this is that these tools suffer from state space explosion and the STGs generated from BALSA can become quite large. A solution can be to decompose a large STG into components that together show the same behaviour and, in the past, we have developed the tool DesiJ for this.

From a BALSA programme, the BALSA compiler produces so-called Breeze component net-lists. In [3, 4], we have described how we can extract the control part of each of these Breeze components and put these parts together, resulting in a large STG. We assume that the reader has quite some acquaintance with this paper. So far, our decomposition very often produced components that were not synthesizable due to so-called irreducible CSC-conflicts. This report collects working notes regarding solutions to this problem. We also document ideas that, in the end, have not helped us to overcome the problem; they might still be useful if developed further.

1.1 Common cause partition

One serious obstacle for dividing a single large STG into smaller components is deciding which signals to keep together in the same component. If we have only few signals in one component and hide all the others, we might lose too much information; as a result, irreducible Complete State Coding (CSC) conflicts can occur and it is impossible to synthesize some signals. A conflict is irreducible if the solver cannot find a way to insert a new internal signal so that the conflict disappears. To understand this, recall that insertion of a new transition before an input transition is not allowed since the environment does not know that it should wait for the new signal before providing the input.

[1]This research was supported by DFG-project 'Optacon' VO 615/10-1.

A common example of irreducible CSC conflicts are self-triggers. Say, there is an input signal a and two output signals x, y that are sequentially connected in a model: $a+ \to x+ \to a- \to y+$. When x and y are put into different components, the component with y will have a self-trigger: $a+ \to a- \to y+$. Here, we have a CSC conflict because the coding information provided by the signal a alone is not sufficient for y to uniquely identify the moment when it should change its value: if $a = 0$, either $y+$ or $a+$ is allowed, but not both. This conflict is also irreducible because signal a is an input and inserting new transitions before $a-$ is not allowed. This leads to the intuition that avoiding such sequential input transitions should help avoiding many if not most irreducible CSC conflicts.

With our current construction, all Balsa-STGs have a specific structure. In particular, no HS-STG has input transitions connected sequentially (with the exception of the arbitration component). The parallel composition of such STGs preserves this property, and thus it is also preserved in the Balsa-STG. The idea is to construct components that preserve this quality. For this, the *common cause partition* puts two output signals in the same part (component) of the output partition if they have a common trigger signal (either input or output).

At first, we also added each trigger that is an output itself to the part, i.e. as an output. In other words, an output o is placed in the same part as another output that triggers o or is triggered by o. The hope is that this partition, just like the "Roughest partition", does not create new irreducible conflicts in the components because all signals preserve their signature (none of the output transitions is converted into an input transition). This *original* (version of) common cause partition easily resulted in at least one component being too large to be synthesizable (here understood to mean that all CSC-conflicts can be solved). Hence, we now define it without this additional placement rule.

Still, the decomposition according to common cause partition usually results in one large part (or very few), a number of "primitive" one-output parts and a few others. With *primitive* we mean that the initial component has a specific structure: for instance, there is just one input signal a and one output signal x, and a is always followed by x, i.e. the postset of each $a+$- ($a-$-)transition consists of a single MG-place forming the preset of an $x+$- ($x-$-)transition. Also, almost automatically, the preset of each $x+$- ($x-$-)transition consists of a single MG-place forming the postset of an $a+$- ($a-$-)transition. Without any reduction of the component (which could lead to additional signals due to backtracking), it is clear that this component can be synthesized as a buffer (or wire), since the signal a alone provides a minimal and complete state coding for x and is sufficient for its implementation.

Similarly, if $x+$ and $x-$ are interchanged, we have an inverter. There are also cases with more than one input and output. As an example, we describe this for an initial component that (presumably) can be synthesized as a combination of two C-elements without any reduction. There are two input signals a and b and two output signals x and y. Signals a and b together are always followed by x and y: the non-dummy transitions always occur in groups of one transition for each of $a+$, $b+$, $x+$ and $y+$, or one for each of $a-$, $b-$, $x-$ and $y-$. In such a group, each of the $a+$- and the $b+$-transition is connected to each of the $x+$- and the $y+$-transition with an MG-place as above, and these places encompass the respective post- and presets, and similarly for $a-$, $b-$ etc.

In principle, there could be signals that can choose between different groups and are lambdarized in the initial component, and it could happen that the $a+$-transition of one group and the $b+$-transition of another group are chosen and fired. Then, a deadlock is reached, while the C-elements erroneously perform $x+$ and $y+$. It is hard to believe that this ever happens in practice, but so far we have not found a water-proof argument against this.

As an example, consider the Balsa-STG GCD in Figure 1. The common cause partition produces 11 components – since an output never triggers another output, the two versions give the same result. There is one large component with output signals $r2$, $r3$, $r4$, rC_11, rD_10, rD_16 (initial component shown in Figure 2), two smaller size components with the outputs rD_21, rS_1 and rD_26, rS_3, and 8 primitive ones with one output each. As an aside, note an interesting property of the large component: if we lambdarize some outputs and the trigger signals not needed for the remaining outputs, this always leads to a self-trigger. In other words, this component cannot be split into smaller components.

The GCD example shows an effect that is expected generally: if we solve the CSC conflicts in the separate

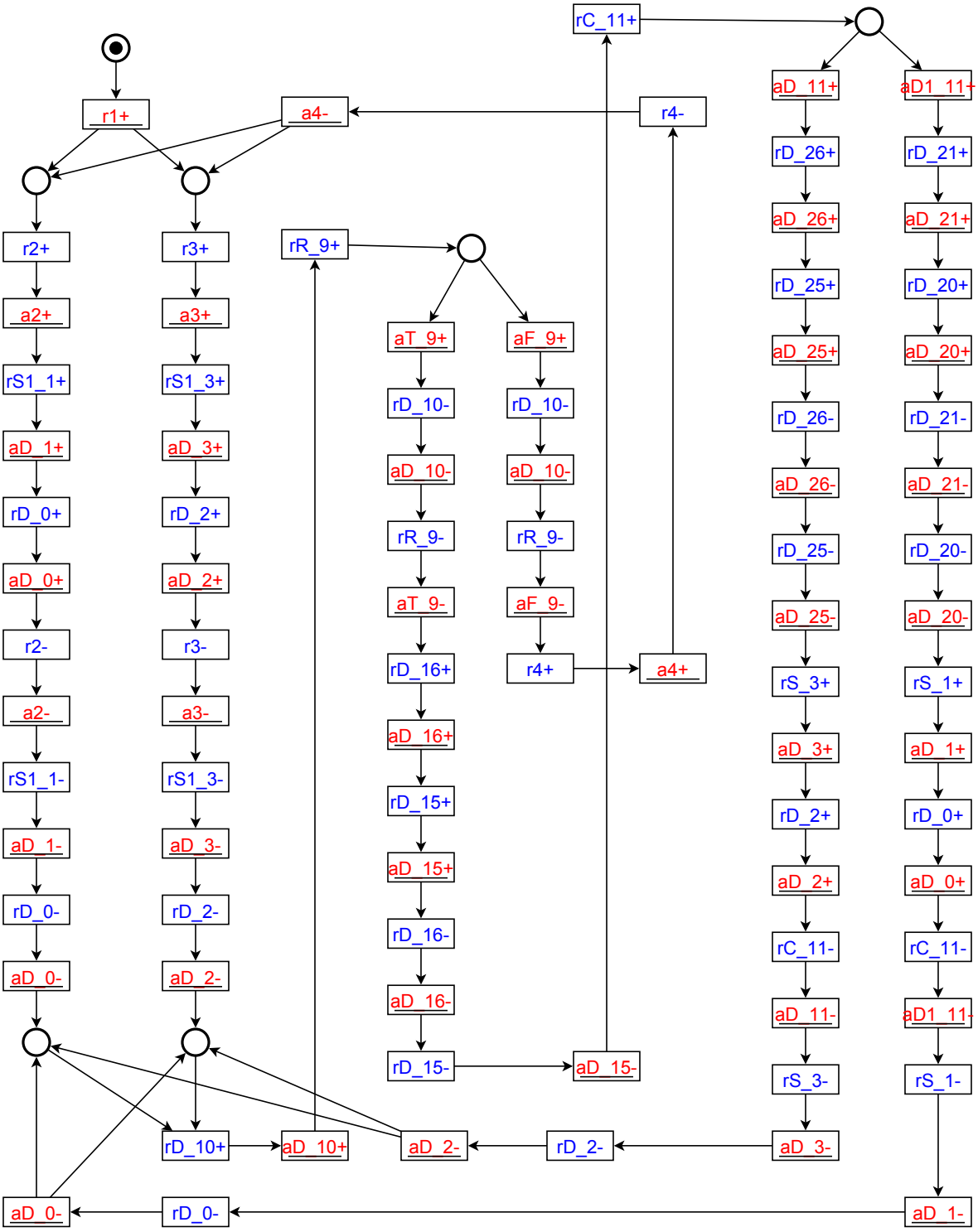

Figure 1: Balsa-STG for the GCD example

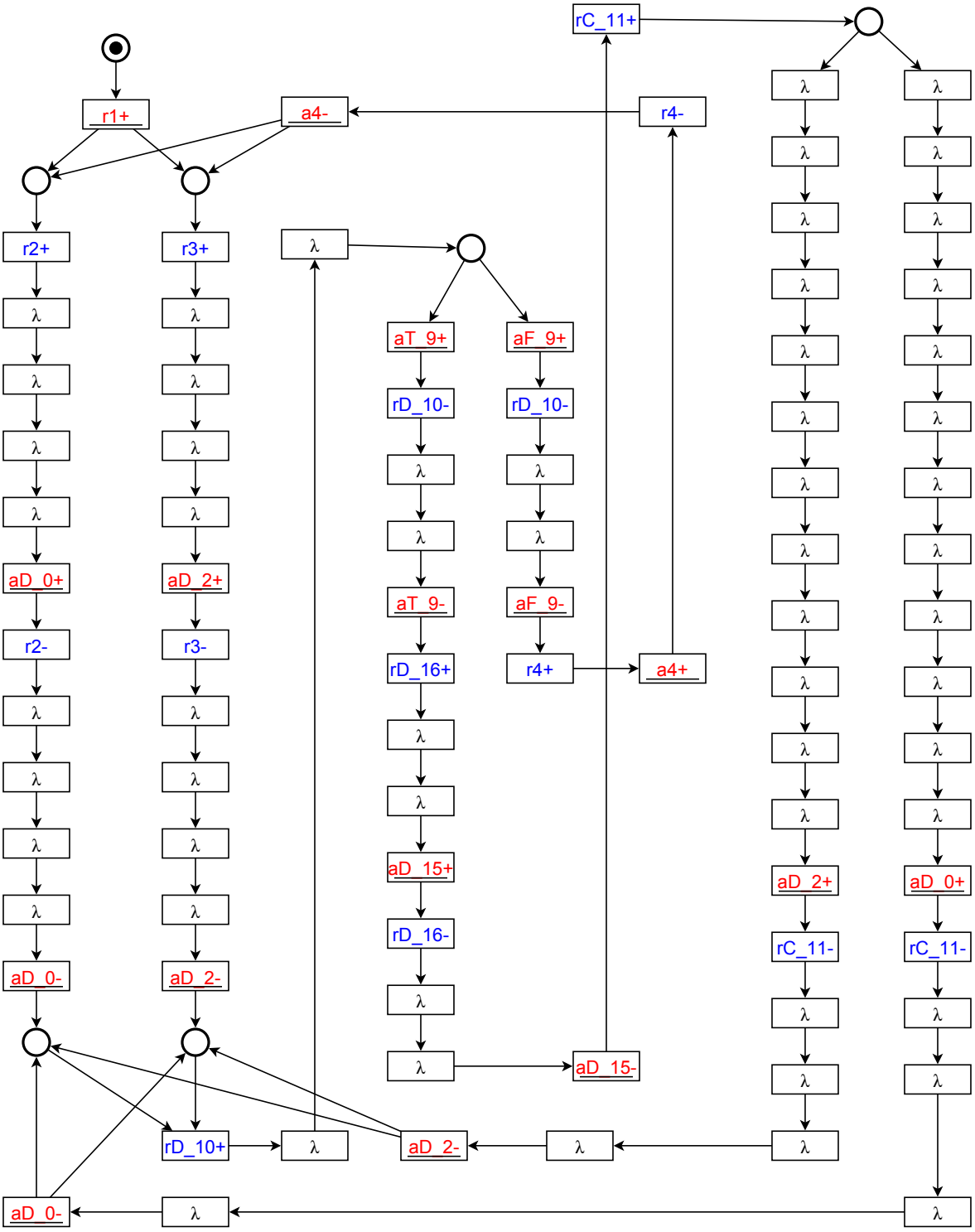

Figure 2: Largest component

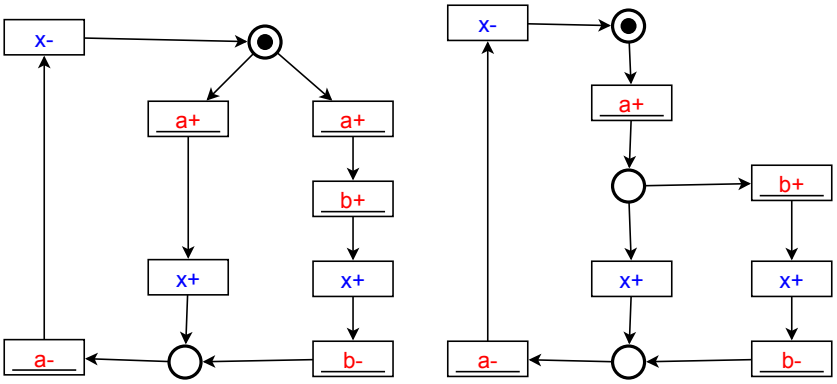

Figure 3: Issue of output-determinacy

components, we need overall more signals than needed if we solve conflicts directly for GCD, using the complete information. For instance, the CSC signals inserted into the largest component can also be used to resolve the conflicts in the two smaller size components. But solving CSC conflicts in the separate components takes less time (where the latter differs significantly between MPSAT and Petrify). It remains to be seen how the performance of the resulting circuits differ.

2 Experiments with the Common Cause partition

Tables 1 to 3 treat a number of Balsa-STG examples in varying ways; for each STG, the number of places was nearly the same as the number of transitions (listed in the second column), and the number of arcs was linear with a small factor. Next, the times for synthesis (i.e. CSC solving) by MPSAT are shown (if possible at all), and the same values are given for the largest component. The data in Tables 1 and 2 were obtained with the original common cause partition, while the data in Table 3 and later tables were obtained with the (improved) common cause partition.

When constructing these Balsa-STGs according to [3, 4] only safe transition contractions are used. Partly for that reason, in each case except GCD, actually a few dummy transitions remained. One can still apply DesiJ in the output-determinate variant [5], but the times for determining the Boolean equations for the Balsa-STG and the largest component were rather large; Table 1 shows some of these times and gives the numbers of remaining dummy transitions in brackets in the second column. Also, for reducing the largest component, only safe transition contractions are used and a number of dummy transitions remain (numbers given in brackets in column 4).

The output-determinate decomposition variant requires that the resulting components be checked for output-determinacy; only in the positive case, the decomposition is guaranteed to be correct. Although, according to the definition, a violation would also be a CSC conflict, it is not clear whether Petrify or MPSAT notices this: they really expect a deterministic STG, and it is not clear how they turn a nondeterministic STG into a deterministic one. E.g., in principle, they might treat the two STGs in Figure 3 in the same way, where only the left one violates output-determinacy, while the right one has an input-output conflict but nevertheless satisfies output-persistence. Luckily, Petrify and MPSAT report the left one to have an irreducible CSC conflict, while they produce Boolean equations for the right one. Thus, we can assume that the latter implies the absence of output-determinacy.

Example	Balsa-STG	MPSAT time	Largest comp.	MPSAT time
GCD	89	134s	31	0.63s
BMU	128 (15)	mem. overflow	96 (15)	mem. overflow
GlobalWinner	134 (1)	8s	94 (1)	4.93s
History Unit	1318 (18)	punf timed out	881 (110)	punf timed out
Arb1	49 (2)	23s (Petrify*)	29 (2)	4s (Petrify)
Arb2	121 (4)	590s	73 (4)	51s
Shift	242 (21)	timed out (>30m)	231 (21)	623s
AAU	341 (8)		137 (8)	750s
MEM	438 (7)		174 (5)	timed out (>30m)
CP0	1119(2)	mem. overflow	355(2)	mem. overflow
DeCode	1234(5)		415(5)	
RegBank	1889(7)	mem. overflow	486 (7)	
EX	1581 (9)		586 (9)	mem. overflow

*MPSAT failed at solving CSC, so Petrify was used instead

Table 1: resolving CSC conflicts in Balsa-STG and the largest component (original Common Cause partition), using Punf and MPSAT

Nevertheless, due to the large synthesis times, a new feature was added to Balsa-STG construction similar to backtracking in decomposition: the remaining dummy transitions were labelled with internal signals in the parallel composition of Breeze-STGs; in a second run of Balsa-STG construction, these signals are not lambdarized but kept (and treated much like outputs). After this modification, the reduction is repeated for the initial Balsa-STG. If again some (new) dummy transitions remain, this backtracking is repeated — and this actually happened with our examples. Table 2 shows some of the results for this approach based on the original version of common cause partition. Naturally, the Balsa-STGs do not have any dummy transitions anymore, but neither the largest components have, except for the History Unit and the EX modules (numbers in brackets). The significance of the modified Balsa-STG construction lies not really in the MPSAT times, but rather in the reduced number of dummy transitions in the components; we believe that this is an important step for enabling us to deal with the large examples having more than 100 transitions; see below.

Finally, Table 3 gives in columns 5-6 the results for the largest components when the real common cause partition in combination with the recovery of internal signals is used. For comparison, two columns of Table 2 are repeated.

In the past, decomposition of practical STGs almost in each case produced at least one component that was not synthesizable. With common cause partition, we have some definite improvement: for GCD, GlobalWinner, Arb1 all components are synthesizable.

2.1 Trying to use unsafe contractions

In the largest component of the Balsa-STG History Unit, there are 75 dummy transitions left uncontracted. One may consider to contract them all using unsafe contractions, which in this example leads to a problem. A fragment of the STG is shown in Figure 2.1. As we contract dummy transitions from left to the right, the number of places in the postset of rE_211+ will double for each contraction even with the redundancy checks in place. As a result, the total number of places for this STG will be more than 2^{16}.

Another interesting observation is that this example contains the same signal transition repeated many times such that the structure itself seems to allow splitting rE_211- as it forms a common path for its predecessors.

Example	Balsa-STG	MPSAT time	Largest comp.	MPSAT time
GCD	89	137s	31	0.66s
BMU	131	3928s	99	3401s
GlobalWinner	135	4.61s	95 —	3.11s
History Unit	1320	punf timed out	625 (63)	punf timed out
Arb1	44	0.24s	27	0.03s
Arb2	125	37s	77	5s
Shift	241	mem. overflow	126	451s
AAU	344		122	637s
MEM	444	timed out (>4h)	178	timed out (>24h)
CP0	1121	mem. overflow	357	mem. overflow
DeCode	1236	mem. overflow	417	mem. overflow
RegBank	1895	mem. overflow	492	mem. overflow
EX	1587	mem. overflow	590 (2)	mem. overflow

Table 2: resolving CSC conflicts in Balsa-STG and the largest component (original Common Cause partition with dummy recovery), using Punf and MPSAT

Example	Balsa-STG	Largest comp. (old)	MPSAT time	Largest comp. (improved)	MPSAT time
GCD	89	31	0.66s	31	0.61s
BMU	131	99	3401s	86	mem. overflow
GlobalWinner	135	95	3.11s	66	0.91s
History Unit	1320	625 (63)	punf timed out	603 (75)	punf timed out
Arb1	44	27	0.03s	27	0.03s
Arb2	125	77	5s	61	1.52s
Shift	241	126	451s	124 (9)	39.75s
AAU	344	140	637s	122	139s
MEM	444	178	timed out (>24h)	162	timed out (>12h)
CP0	1121	357	mem. overflow	351	timed out (>4h)
DeCode	1236	417	mem. overflow	371	timed out (>4h)
RegBank	1895	492	mem. overflow	472	mem. overflow
EX	1587	590 (2)	mem. overflow	574 (2)	mem. overflow

Table 3: resolving CSC conflicts, (improved) Common Cause partition

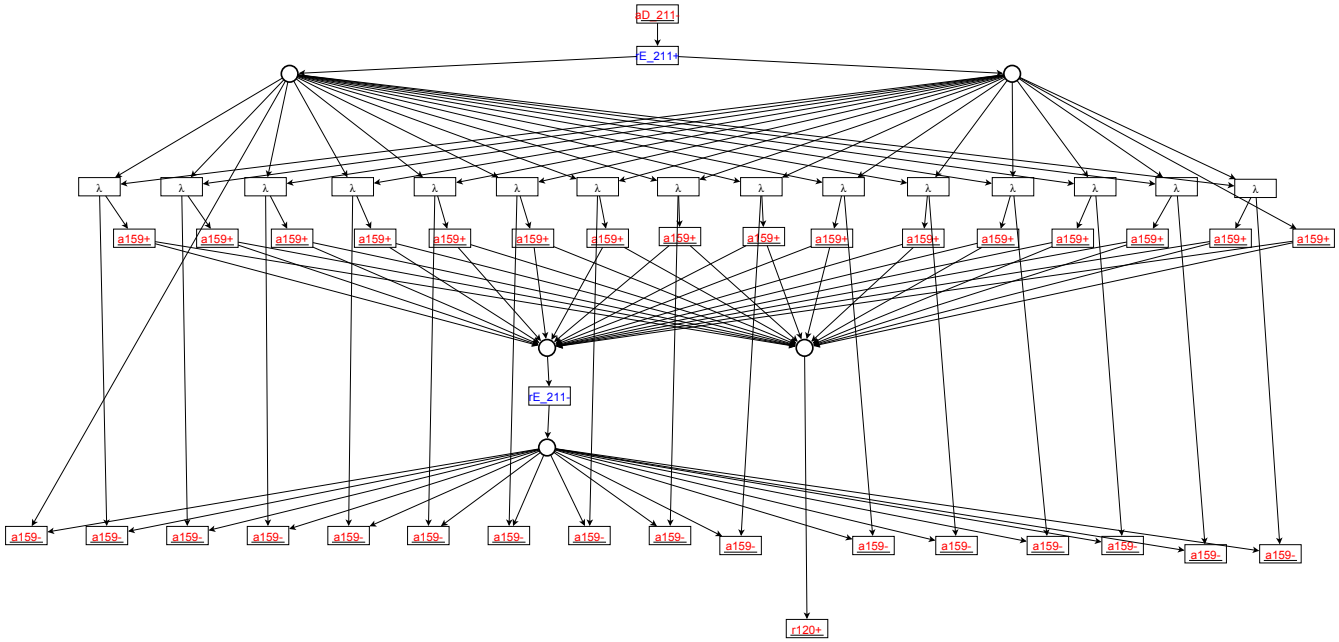

Figure 4: STG fragment of the History Unit module

Perhaps some more sophisticated algorithms of path splitting and place merging could help contracting all the remaining dummy transitions while only relying on safeness preserving contractions.

2.2 "Sandwich" partition heuristics

This heuristic didn't really help for the Balsa examples (it is more time consuming and the largest component is even larger), but preliminary experiments (not shown) indicate that it is helpful when dealing with sequences of input transitions.

In previous examples we have shown an easy way to detect a great amount of "primitive" signals (output signals that can be generated with their direct triggers only) using the common cause partition. Another interesting question is: how many other signals are there that can be implemented while knowing only their syntactic triggers? When creating such a "level 1" component, we turn all of the syntactic triggers of some signal x into inputs. Only the signal in question is an output or an internal signal. The rest of the transitions are lambdarized and contracted. If MPSAT can solve all the CSC conflicts, this output signal has enough information and it is implementable. If MPSAT cannot find the solution, the chances are that some of the triggers have to be restricted by adding new arcs and transitions. This becomes the subject for the SW partition (see below). One may also wonder about creating the "level 2" component, where the second layer of triggers is added as well, i.e. the syntactic triggers of the syntactic triggers are converted into inputs and added to form the component in question. This component shows whether a signal can be implemented without constraining its triggers while knowing behaviour of an extended number of triggers.

The SW component is a special STG, which is generated from a "level 1" component for some chosen signal x and splitting each trigger (necessarily an input) into a pair of an input followed by a (new) output. The new input (output) transition inherits the preset (postset) of the trigger transition, and an empty marked graph place

Example	Output signals	Common cause	Level 1	Level 2	solved SW	Partially solved SW	Unsolved
GCD	18	8	0	0	0	10	0
BMU	37	8	6	0	0	23	0
GlobalWinner	20	4	4	0	0	12	0
HistoryUnit	252	193	very slow				59
Arb1	9	3	2	0	0	0	4
Arb2	27	6	8	0	0	7	6
Shift	60	27	4	0	0	28	1
AAU	70	26	6	0	0	8	30
MEM	89	41	7	0	0	34	7
CP0	186	110	11	0	0	38	27
DeCode	166	86	7	0	0	67	6
RegBank	299	203	22	0	0	68	6
EX	294	135	37	0	0	66	56

Table 4: Number of signals implementable with only direct triggers

is connecting input to the output. If all CSC conflicts are solved, we check whether this added a new trigger to one of the new outputs. If this is the case, it indicates that the trigger a the output was created from would need a new CSC signal as a trigger. If a is an output, we should add it as an output to the component of x to be able to restrict it with a CSC signal; if a is an input, we would have to add at least one of its triggers in turn. In any case, x is not primitive; if the check fails, x is not primitive.

If the SW component is not solved for some signal, this means that MPSAT took very long to solve it, or it failed with an error message that the CSC conflicts cannot be resolved (this doesn't mean the CSC is irresolvable, as on not very big examples Petrify was usually able to find the solution).

The Table 4 shows the total number of signals in the initial STG, then the primitive signals found by the common cause partition, then the additional primitive signals found by the new method.

According to this table, if a signal was not separated by the common cause partition (which usually happens with C-elements, inverters, and buffers), the chances are that this signal is not implementable with only it's syntactic triggers.

3 Effect of changing FalseVariable definition

The following table presents the effect of changing the definition of the FalseVariable component. The idea is to remove some of the arcs, which are expected to be ensured by the environment of the component, and to get a smaller Balsa-STG this way.

Old definition:

$$\begin{aligned} \mathit{scaled} \quad & C \\ \mathit{active} \quad & B \\ f \quad = \quad & \#(A : (rB+; \#||(\#C); aB+; \bigtriangledown B))) \end{aligned}$$

New definition (the expression features separated channel C and TELEM instead of SELEM):

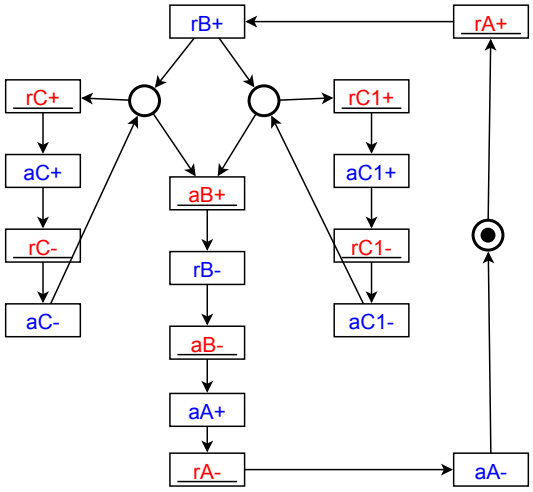

Figure 5: Original FalseVariable

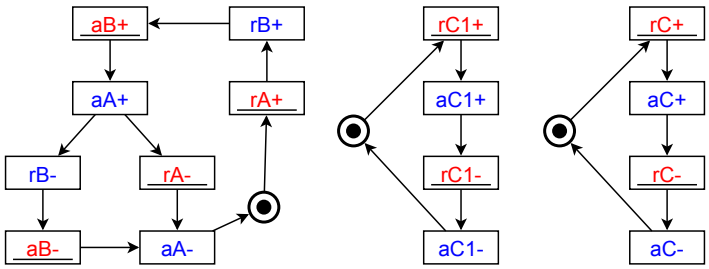

Figure 6: FalseVariable with TELEM and separated channels

$$
\begin{aligned}
\mathit{scaled} \quad & C \\
\mathit{active} \quad & B \\
f \quad = \quad & \#(rA+;rB+;aB+;aA+;((rB-;aB-)||rA-);aA-)||(\#||(\#(C)))
\end{aligned}
$$

New definition 2 (the expression features separated channel C and SELEM):

$$
\begin{aligned}
\mathit{scaled} \quad & C \\
\mathit{active} \quad & B \\
f \quad = \quad & \#(A:(\triangle B;\triangledown B))||(\#||(\#(C)))
\end{aligned}
$$

The experiment has shown that separating the channel C from the component has no visible effect (the same amount of arcs and transitions). Presumably the main reason for such a result is that optimized parallel composition and LP-solver based redundant place removal are efficient enough and can automatically remove these arcs. A

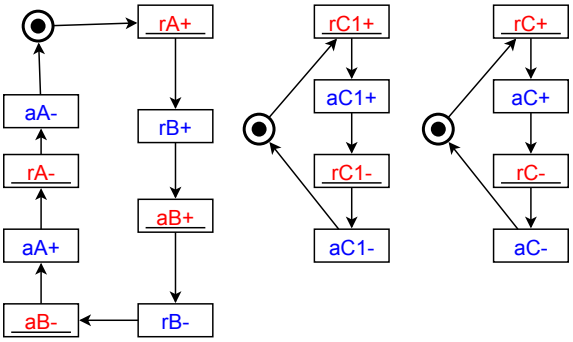

Figure 7: FalseVariable with SELEM and separated channels

	old		new		new 2	
Example	#arcs	#transitions	#arcs	#transitions	#arcs	#transitions
GCD	183	89	183	89	183	89
BMU	351	131	351	131	351	131
GlobalWinner	295	135	295	135	295	135
HistoryUnit	2897	1320	2922	1260	2897	1320
Arb1	104	44	108	44	104	44
Arb2	286	125	290	125	286	125
Shift	640	241	696	221	640	241
AAU	743	331	764	336	743	331
MEM	950	444	954	444	950	444
CP0	2374	1121	2446	1098	2374	1121
DeCode	2581	1236	2587	1225	2581	1236
RegBank	3960	1895	3999	1864	3960	1895
EX	3443	1587	3470	1589	3443	1587

Table 5: Effect of changing the definition of the FalseVariable component

rather surprising result concerns the FalseVariable containing a TELEM component. It usually causes more arcs and reduces the number of transitions in the STG. It is not quite clear, how this can be explained.

In the end, it will be important how the different versions support reduction operations, hopefully leading to smaller components.

4 Avoiding large STG components when the "Common cause partition" is used

It can be observed that using the common cause partition on the Balsa-STG with *all* internal signals present usually produces small components, but of course we want to get rid of most of them. Here we develop a method that estimates, which internal signals to keep (avoid lambdarizing) so that the common cause partition does not end up with large components. The method works as follows:

Example	prev. Balsa-STG	Balsa-STG	Largest comp.	its MPSAT time
GCD	89	103	21	<1s
BMU	131	239	36	9.16s
GlobalWinner	135	187	51 (9)	<1s
History Unit	1320	timed out		
Arb1	44	65	23	<1s
Arb2	125	145	32	<1s
Shift	241	459	101 (35)	mem. overflow
AAU	344	433	37 (5)	<1s
MEM	444	540	40 (4)	<1s
CP0	1121	1472	139 (11)	slow
DeCode	1236	1919	384	<1s
RegBank	1895	2383	146 (2)	<1s
EX	1587	2232	113 (15)	<1s

Table 6: Avoiding large components (number of transitions for Balsa-STGs and new largest components)

- Compose Balsa-STG while keeping the internal communication signals
- For each internal signal s that was not lambdarized or marked as problematic (see below)), we do the following:
  - $Tr := \emptyset$; $E := \emptyset$ (initializes the sets of "trigger signals" and "effect signals")
  - Add s to both Tr and E, and accumulate all related signals the way it happens with the common cause partition: repeatedly add *all* signals triggered by some $r \in Tr$ to E (at the moment, this includes also input signals – this should be reconsidered) and *all* signals triggering some $e \in E$ (where e is not an input signal) to Tr; for a λ-transition t triggered by some $r \in T$, determine the "iterated" post-postset (iterated over λ-transitions) and add its signals to E, and analogously for a λ-transition triggering some $e \in E$. (In more detail, the iterated post-postset of t is determined as follows: we start from set $\{t\}$ and repeatedly add the post-postset of each λ-transition in the set until the construction stablizes.)
  - Let set C consist of all internal signals that occur in both Tr and E (we have s in C, but C may be larger).
  - if $|(Tr \cup E) \backslash C|$ is smaller than some threshold value (and also the number of input signals and the number of output signals in this set are smaller than some threshold value), then lambdarize all signals in C that are not marked as problematic.
- Contract lambdarized signals
- An additional step: if after contraction, there are some dummy transitions left, mark their signals as problematic; these signals will never be lambdarized. Then repeat the procedure for the initial Balsa-STG. This step ensures that there are no dummy transitions left in the end.

The idea for the loop over the signals s is as follows. If we lambdarize and contract s, the triggers of s will become triggers of the effects of s; in essence, the former are added to Tr, the latter to E and then we continue to determine a set of local signals according to common cause partition.

Table 6 shows the effect of using this method (note the increased count of transitions in the Balsa-STG):

In this approach, there are usually a number of large components. Perhaps, all of these should be tested here (not just the single largest one).

4.1 Problematic STGs

The component that ended with memory overflow is shown in Figure 8. Luckily, some of these dummy transitions can be removed by transition merging as described in the next subsection.

4.2 The transition merge procedure (the "zip up" operation)

This is a structural STG transformation technique, which helps to remove more dummy transitions. It starts from a merge place, i.e. a place with more than one transition in its preset and tries to merge these transitions. Only transitions with the same postset and the label can be merged in order to decrease the number transitions. Two transitions of the same signal edge (or two dummy transitions) t_1, t_2 can be merged, if the following conditions are true:

- t_1 and t_2 have a common postset: $\forall p \in t_1^\bullet \cup t_2^\bullet : W(t_1, p) = W(t_2, p)$
- there is exactly one place $p_1 \in {}^\bullet t_1$, which is not containted in ${}^\bullet t_2$
- there is exactly one place $p_2 \in {}^\bullet t_2$, which is not containted in ${}^\bullet t_1$
- t_1 and t_2 have a common preset apart from p_1 and p_2: $\forall p \in {}^\bullet t_1 \cap {}^\bullet t_2 : W(p, t_1) = W(p, t_2)$
- $W(p_1, t_1) = W(p_2, t_2) = 1$ and $|p_1^\bullet| = |p_2^\bullet| = 1$

Merging of t_1 and t_2 also merges p_1 and p_2 as follows:

1. $\forall t \in {}^\bullet p_2 : W(t, p_1) := W(t, p_1) + W(t, p_2)$
2. $M(p_1) := M(p_1) + M(p_2)$
3. remove p_2 and t_2 from the model

Additional note:

This operation is potentially in conflict with the merge-place splitting operation because they do the opposite operations and might stall the programm in an endless loop. One could disable merging if there are dummy transitions in ${}^{\bullet\bullet} t_1 \cup {}^\bullet t_2$ or disable the merge-place splitting. Merging dummy transitions does not create endless loop because splitting dummies is not allowed.

4.2.1 Using the improved merge procedure

We used improved merge in the generation of the Balsa-STG and in the reduction of components; in turn, we stopped using the merge-place splitting operation. This explains the slightly larger number of transitions in the last example in Table 7. The modified strategy for reducing an STG with dummies is shown in Figure 9, cf. [4]; improved merge is added at the end of the main loop.

After reducing the component in Figure 8 while using the improved merge operation, the result looks as in Figure 10 and can be easily processed with MPSAT.

4.3 Avoiding large STG components with transition merging enabled

Table 7 presents the new estimation of the largest components, when the merging is used. Note that the largest component might now be a different one; this could explain the failure for example EX.

The size of the component in terms of transition is probably not a good estimation for the hardness of synthesis. More component testing should be done here.

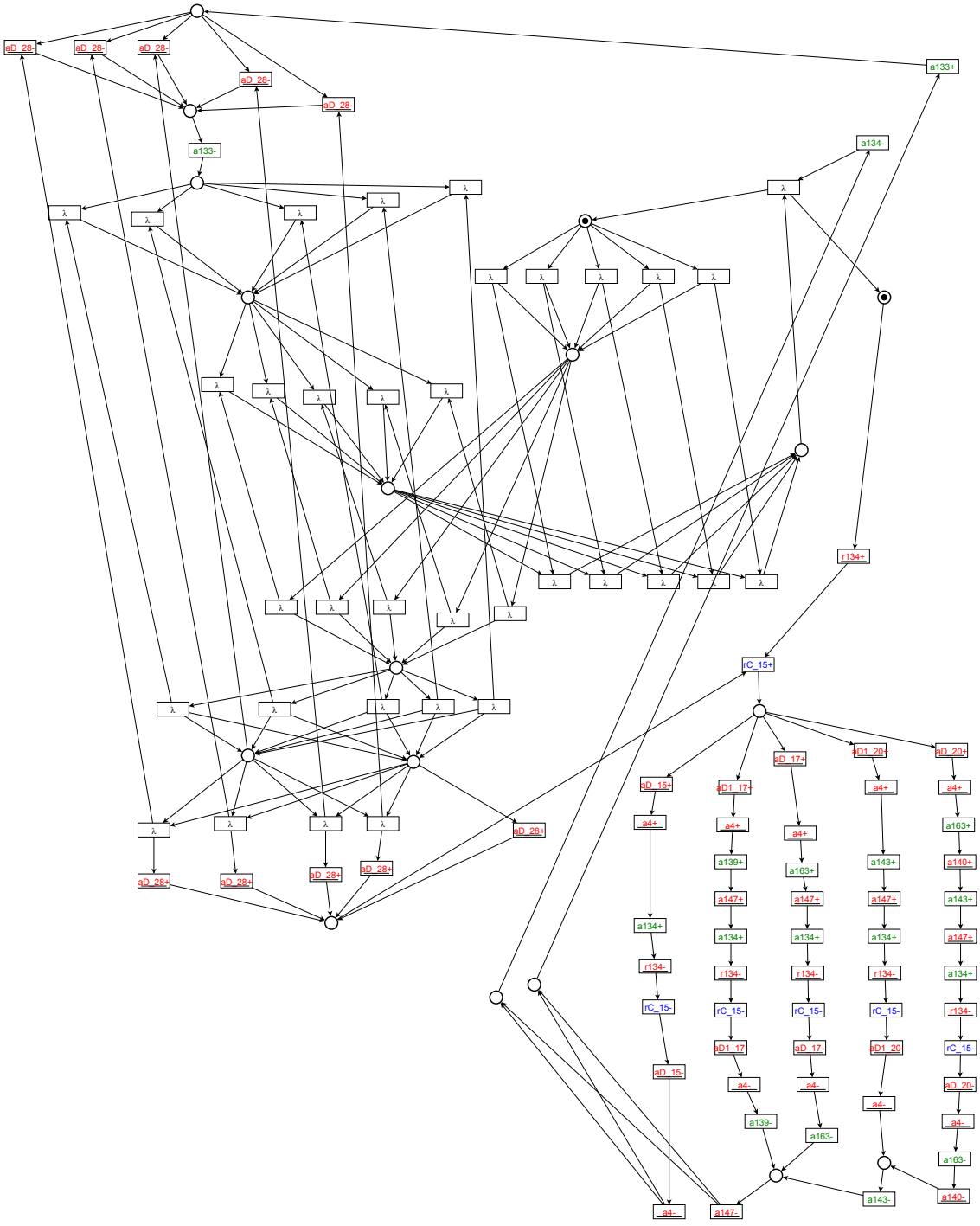

Figure 8: Largest component in module “Shift”

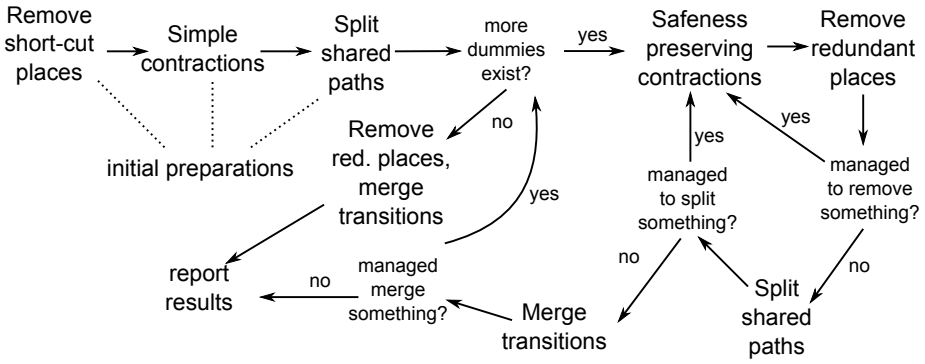

Figure 9: New reduction procedure with merging

Example	prev. Balsa-STG	Balsa-STG	Prev. largest comp. size	Largest comp. size	MPSAT time
GCD	103	103	21	21	<1s
BMU	239	239	36	36	9s
GlobalWinner	187	187	51 (9)	30	<1s
History Unit	timed out	1680	-	49	<1s
Arb1	65	65	23	23	<1s
Arb2	145	145	32	32	<1s
Shift	459	459	101 (35)	59 (1)	<1s
AAU	433	433	37 (5)	29 (1)	<1s
MEM	540	540	40 (4)	40 (4)	<1s
CP0	1472	1472	139 (11)	128	timed out
DeCode	1919	1919	384	384	<1s
RegBank	2383	2383	146 (2)	146 (2)	<1s
EX	2232	2238	113 (15)	100 (2)	failed to solve CSC

Table 7: Avoiding large components with transition merging

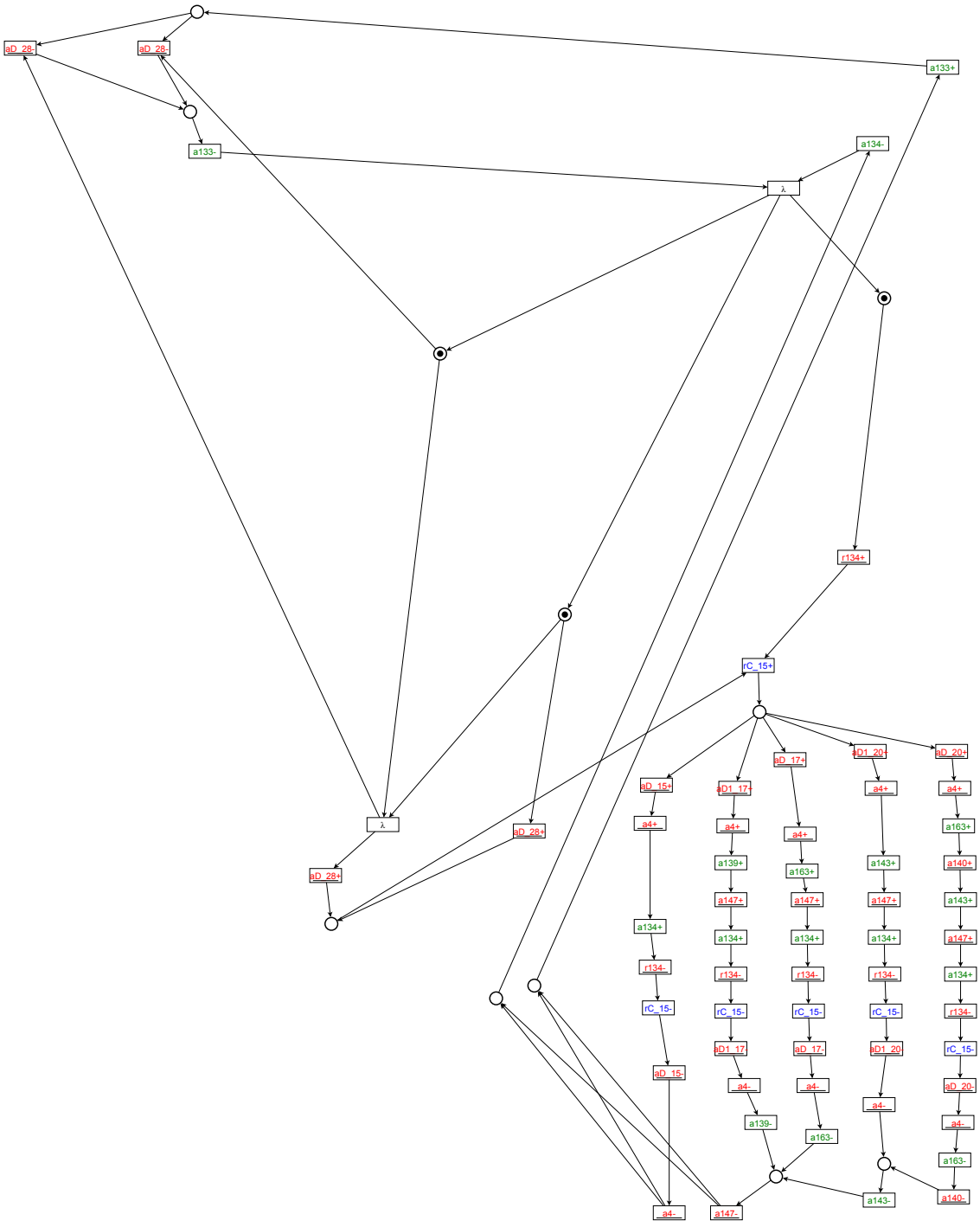

Figure 10: Largest component in module “Shift” after using the improved merge procedure

5 Problem with slow BrzSequence components

The Table 7 shows that the largest component of CP0 timed out while resolving CSC. This slowdown was caused by a long list of simple sequentially connected operations. Apparently, having simple operations governed by one sequencer can cause the inability of the proposed method to generate small components. Another interesting observation is that a BrzSequence component (which is a simple sequence of transitions) scaled to a large degree creates a very difficult task for both Petrify and MPSAT when they solve CSC. Our solution to this problem is to change the definition of BrzSequence so that it has additional internal signals that deal with CSC conflicts. Checking CSC in BrzSequence with these additional signals is very fast regardless of the component size. The new definition uses the wire rC to eliminate CSC conflicts in this component, it is defined as follows:

$$\begin{aligned} &\mathit{active} && B, C \\ &\mathit{scaled} && B, C \\ &f &=\ & \#(rA + .\#.(rB + .aB + .rC + .rB - .aB-).aA + .rA - .\#.(rC-).aA-) \end{aligned}$$

With this change, MPSAT solved the problematic component in less than 1 s.

6 Avoiding irreducible CSC conflicts

When decomposing large STGs into smaller STG components, there are two types of irresolvable CSC conflicts that typically occur when decomposing Balsa-STGs:

1. Self-trigger. When an output transition is lambdarized, it may create a self-trigger (some input transition that by firing in one direction immediately enables its opposite edge). When $a+$ is directly followed by $a-$ and $a-$ drives some output transition $x+$, there is not enough information for x to know when it should fire. This type of conflict is avoided in the Common Cause Partition by design: In all the Balsa-STGs we looked at, there are never two inputs in sequence, and Common Cause Partition adds all the outputs that have the same triggers.

2. Multiple occurrence conflict. This conflict occurs when some input transition has multiple occurrences. For instance, a CSC conflict exists because one occurrence of input $a+$ triggers an output $x+$, another instance of $a+$ does not. Because input transitions cannot be prepended by extra signals, it may be impossible to solve this conflict. Structurally, we detect such a conflict by checking the signals of post-postsets of the two $a+$ transitions for inequality (false positives are not a big problem). More precisely, we try to find a bijection between the places of the postsets such that each place and its related place has the same signals in its resp. postset: if this fails, we have detected a potential conflict.
   To help resolving this conflict, one or more signals are recovered and added to the component as inputs, so that output signal $x+$ can distinguish the different $a+$ occurrences. For each problematic trigger $a+$, find one signal to recover as follows:

   (a) Starting from $a+$, iteratively gather the transitions in the pre-presets until either a choice-place transition is met or the opposite edge of the problematic trigger $a-$ is met or there are no more transitions to add.

   (b) Among the transitions gathered, choose one transition such that its signal edge is unique in the whole STG and the opposite edge is not represented in the gathered set; in Balsa-STGs, such unique edge occurrences are quite common. Add the signal found as a new input, the uniqueness will guarantee that different $a+$ can be distinguished.

Example	prev. Balsa-STG	Balsa-STG	Prev. largest comp. size	Largest comp. size	all solved?	all synthesized?
GCD	103	103	21	25 (1)	yes	
BMU	239	239	36	36	yes	
GlobalWinner	187	187	30	32	yes	
History Unit	1680	1680	49	49	yes	
Arb1	65	65	23	23	yes	
Arb2	145	145	32	44	yes	
Shift	459	459	59 (1)	61 (1)	yes	
AAU	433	433	29 (1)	29 (1)	yes	
MEM	540	540	40 (4)	44 (4)	yes	
CP0	1472	1472	128	128	yes	
DeCode	1919	1919	384	390	yes	
RegBank	2383	2383	146 (2)	148 (2)	yes	
EX	2238	2238	100 (2)	116 (2)	yes	

Table 8: Avoiding large components and avoiding irr. CSC conflicts

(c) If no such signal is found, try using the signal of a choice-place transition as a solution if such a transition was found (even if this is not a unique occurrence, there are usually good chances that the choice made before the problematic trigger will help to resolve the CSC conflict).

Refinements of this approach are possible in order to add fewer signals. The result of adding this CSC avoidance is shown in Table 8. It was possible to resolve all CSC conflicts in all components for each design.

7 Examples of remaining dummy transitions inside components

The following figures demonstrate examples where dummy transitions failed to contract so far. In some cases, one can define new admissible (even safeness-preserving) operations to the decomposition approach that remove all dummy transitions in one go.

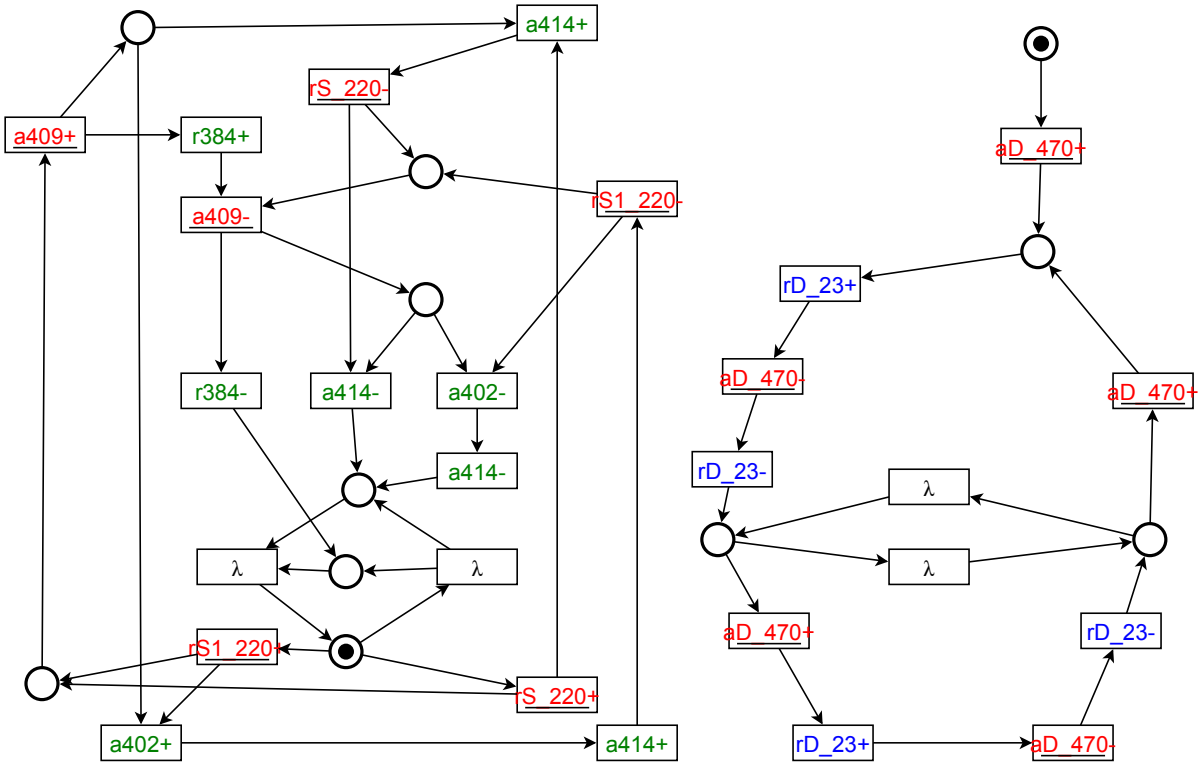

Figure 11: Failed to contract

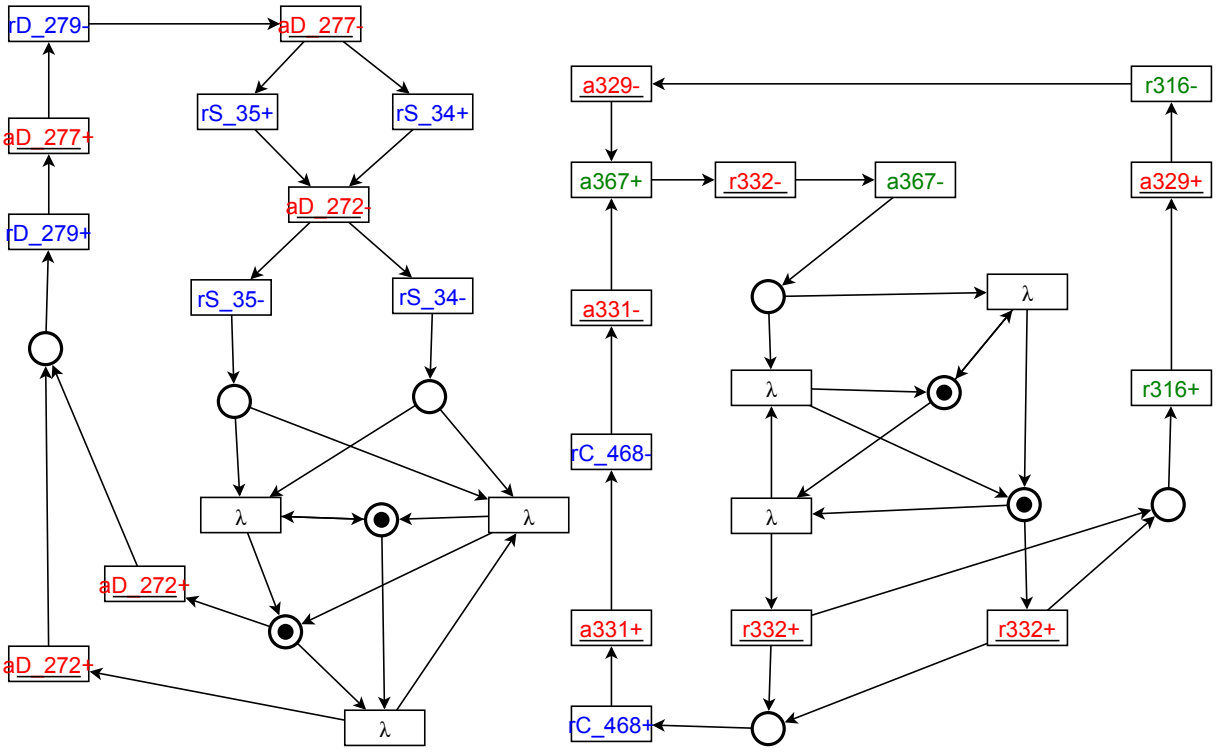

Figure 12: Failed to contract

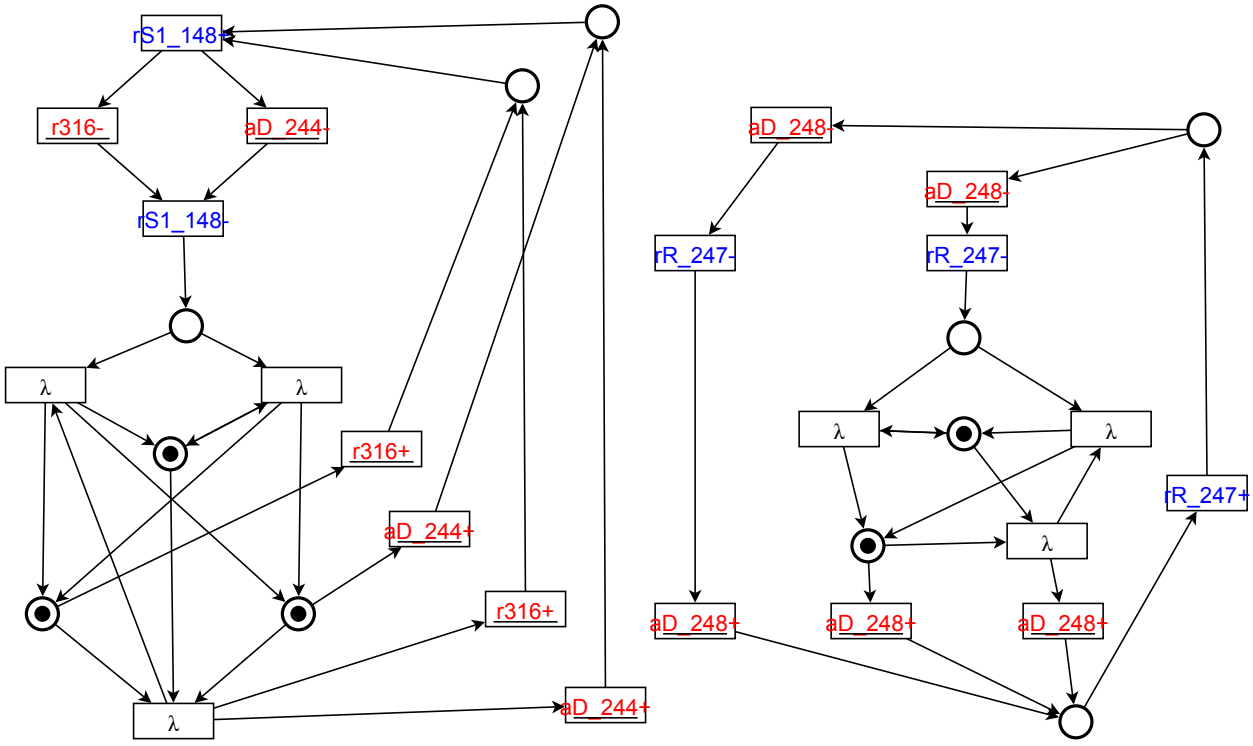

Figure 13: Failed to contract

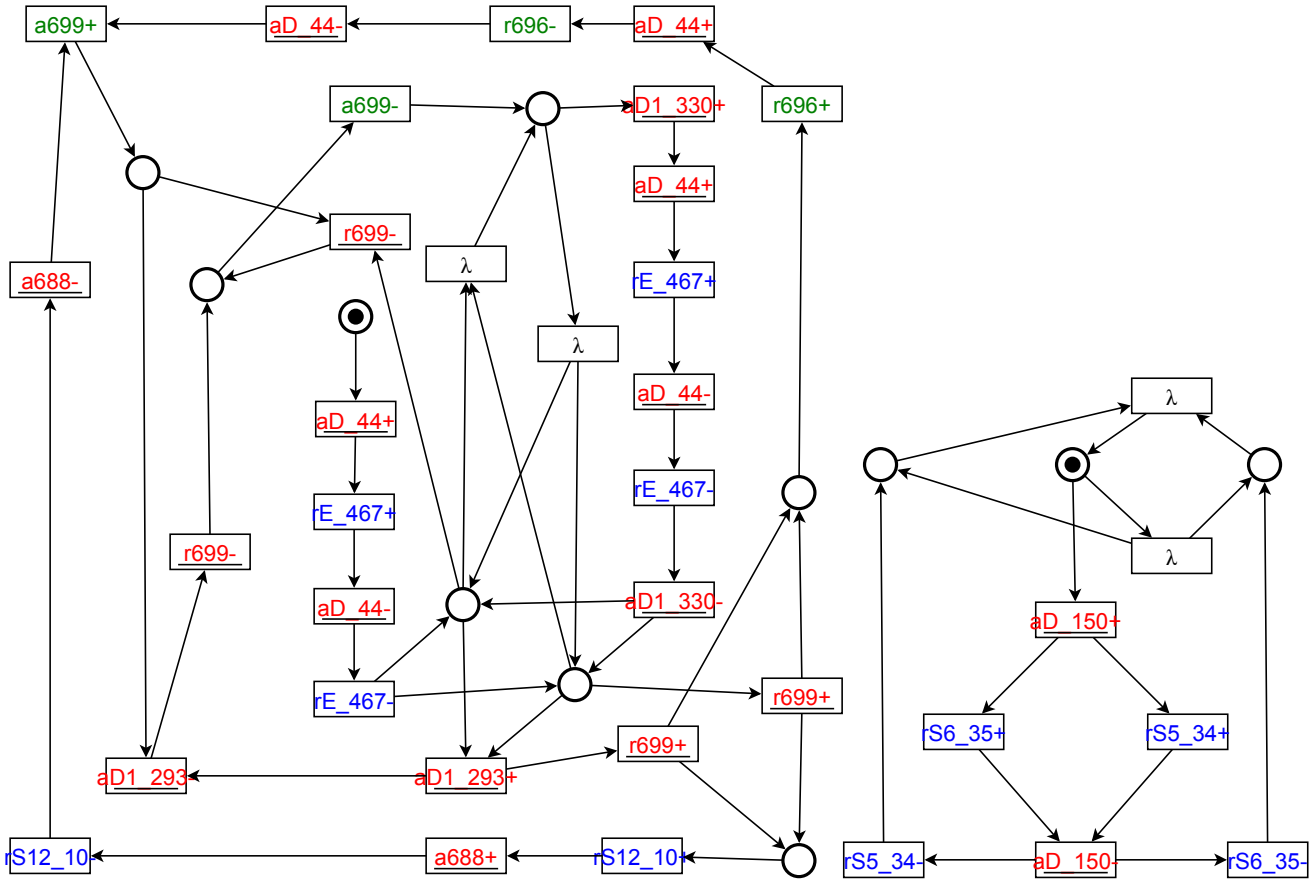

Figure 14: Failed to contract

8 Using DesiJ

This section shows examples on how to process breeze files:

- To launch DesiJ in GUI mode, use:

```
$ desij.sh -G
```

- To convert a breeze file to the Balsa-STG, use:

```
$ desij.sh -Y -g -t operation=breeze outfile=<output Balsa-STG .g> <input .breeze>
```

-g option ensures channels are preserved as internal signals, and not directly lambdarized
-Y allows risky strategy contractions
-t enables depth-based LP solver (see also option *lp-solver-depth*)
operation=breeze tells desiJ to read the *.breeze* file

- To decompose the Balsa-STG, run:

```
$ desij.sh -Y -t partition=common-cause version=breeze <input Balsa-STG .g>
```

partition=common-cause uses the "Common cause" partition algorithm
version=breeze uses decomposition aimed at STGs created from breeze files. It is mostly similar to the basic decomposition.

- If the program ends up out of memory, try increasing Java's maximum heap size inside the starter scripts. In the following example it is set to 1Gb:

```
java -Xmx1g -jar $DESIJ_DIR/desij.jar $*
```

References

[1] "The Balsa Asynchronous Circuit Synthesis System:
http://www.cs.manchester.ac.uk/apt/projects/tools/balsa/."

[2] A. Bardsley, "Implementing Balsa handshake circuits," Ph.D. dissertation, Department of Computer Science, University of Manchester, 2000.

[3] S. Golubcovs, W. Vogler, and N. Kluge, "Stg-based resynthesis for balsa circuits," in *Application of Concurrency to System Design (ACSD), 2013 13th International Conference on*, 2013, pp. 140–149.

[4] S. G. Walter Vogler Norman Kluge, "Stg-based resynthesis for balsa circuits," Augsburg University, Tech. Rep., 2013.

[5] V. Khomenko, M. Schaefer, and W. Vogler, "Output-determinacy and asynchronous circuit synthesis," *Fundamenta Informaticae*, vol. 88, no. 4, pp. 541–579, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1497088.1497094